CS 202 - Computer Science II

Project 9

Due date (FIXED): Wednesday, 4/24/2019, 11:59 pm

Objectives: The main objectives of this project are to test your ability to create and use queue-based dynamic data structures. A review of your knowledge to manipulate dynamic memory, classes, pointers and iostream to all extents, is also included. You may from now on freely use **square bracket**-indexing, **pointers**, **references**, all **operators**, the **<cstring>** library, and the **std::string** type as you deem proper.

Description:
For this project you will create a Queue class, both an Array-based and a Node-based variant. A Queue is a First In First Out (FIFO) data structure. A Queue exclusively inserts data at the back (**push**) and removes data from the front (**pop**). The Queue's **front** data member points to the first inserted element (the front one), and the **back** data member points to the last (the rear one).

Array-based Queue:
The following ArrayQueue.h file extract is used to explain the required specifications for the class (it implements a Queue handling DataType objects):

```
const size_t ARRAY_MAX = 1000;

class ArrayQueue{
  friend std::ostream & operator<<(std::ostream & os,                    //(i)
                                   const ArrayQueue & arrayQueue);

  public:
    ArrayQueue();                                                         //(1)
    ArrayQueue(size_t count, const DataType & value);                     //(2)
    ArrayQueue(const ArrayQueue & other);                                 //(3)
    ~ArrayQueue();                                                        //(4)

    ArrayQueue & operator= (const ArrayQueue & rhs);                      //(5)

    DataType & front();                                                  //(6a)
    const DataType & front() const;                                      //(6b)

    DataType & back();                                                   //(7a)
    const DataType & back() const;                                       //(7b)

    void push(const DataType & value);                                    //(8)
    void pop();                                                           //(9)

    size_t size() const;                                                 //(10)
    bool empty() const;                                                  //(11)
    bool full() const;                                                   //(12)
    void clear();                                                        //(13)
    void serialize(std::ostream & os) const;                             //(14)

  private:
    DataType m_array[ARRAY_MAX];
    size_t m_front;
    size_t m_back;
    size_t m_size;
};
```

The **ArrayQueue** Class will contain the following **private** data members:

- **m_array,** the array that holds the data. *Note*: This is a Statically Allocated array that holds ARRAY_MAX number of objects, be careful *not* to treat is as a Dynamically Allocated one.
- **m_size**, a size_t, keeps track of how many elements are currently stored in the Queue (and therefore considered valid). *Note*: This cannot exceed ARRAY_MAX.
- **m_front,** a size_t, with the respective m_array index of the front element of the Queue.
- **m_back**, a size_t, with the respective m_array index of the back element of the Queue.

,will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new Queue object with no valid data.
- **(2) Parametrized Constructor** – will instantiate a new Queue object, which will hold size_t count number of elements in total, all of them initialized to be equal to the parameter value.
- **(3) Copy Constructor** – will instantiate a new Queue object which will be a separate copy of the data of the **other** Queue object which is getting copied. *Note*: Consider whether you actually need to implement this.
- **(4) Destructor** – will destroy the instance of the Queue object. *Note*: Consider whether you actually need to implement this.
- **(5) operator=** will assign a new value to the calling Queue object, which will be an exact copy of the rhs object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. *Note*: Consider whether you actually need to implement this.
- **(6a,6b) front** returns a Reference to the front element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.
- **(7a,7b) back** returns a Reference to the back element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.
- **(8) push** inserts at the back of the Queue an element of the given value. *Note*: Since m_size can never exceed ARRAY_MAXSIZE, checking if the Queue is full prior to pushing a new element makes sense.
- **(9) pop** removes from the front element of the Queue. *Note*: Since m_size is an unsigned value, checking if the Queue is empty prior to popping an element makes sense.
- **(10) size** will return the size of the current Queue.
- **(11) empty** will return a bool, true if the Queue is empty (m_size==0).
- **(12) full** will return a bool, true if the Queue is full (m_size==ARRAY_MAXSIZE).
- **(13) clear** performs the necessary actions, so that after its call the Queue will be semantically considered empty.
- **(14) serialize** outputs to the parameter ostream os the complete content of the calling Queue object.

as well as a non-member overload for:

- **(i) operator<<** will output (to terminal or file) the complete content of the arrayQueue object passed as a parameter.

Final Note about Array-based Queue:

The required implementation will be a wrap-around Queue. This means that:

a) Pushing an element will move the back by one (m_back = (m_back+1) % ARRAY_MAXSIZE) as seen in Lecture 20 and increment the m_size by 1.
b) Popping an element will move the front by one (m_front = (m_front+1) % ARRAY_MAXSIZE) as seen in Lecture 20 and decrement the m_size by 1.
c) Keeping track of the count of elements in the Queue (through m_size) as seen in Lecture 20 is necessary.

Node-based Queue:
The following NodeQueue.h file extract is used to explain the required specifications for the class (the corresponding Node class handles DataType objects, as per your previous Project) :

```
class NodeQueue{
  friend std::ostream & operator<<(std::ostream & os,                    //(i)
                                   const NodeQueue & nodeQueue);
  public:
    NodeQueue();                                                          //(1)
    NodeQueue(size_t size, const DataType & value);                       //(2)
    NodeQueue(const NodeQueue & other);                                   //(3)
    ~NodeQueue();                                                         //(4)
    NodeQueue& operator= (const NodeQueue & rhs);                         //(5)
    DataType & front();                                                  //(6a)
    const DataType & front() const;                                      //(6b)
    DataType & back();                                                   //(7a)
    const DataType & back() const;                                       //(7b)
    void push(const DataType & value);                                    //(8)
    void pop();                                                           //(9)
    size_t size() const;                                                //(10)
    bool empty() const;                                                 //(11)
    bool full() const;                                                  //(12)
    void clear();                                                       //(13)
    void serialize(std::ostream & os) const;                            //(14)
  private:
    Node * m_front;
    Node * m_back;
};
```

The **NodeQueue** Class will contain the following **private** data members:

- **m_front,** a Node Pointer type, pointing to the front element of the Queue.
- **m_back**, a Node Pointer type, pointing to the back element of the Queue.

,will have the following **public** member functions:

- **(1) Default Constructor** – will instantiate a new Queue object with no elements (Nodes).
- **(2) Parametrized Constructor** – will instantiate a new Queue object, which will dynamically allocate at instantiation to hold size_t count number of elements (Nodes), all of them initialized to be equal to the parameter value.
- **(3) Copy Constructor** – will instantiate a new Queue object which will be a separate copy of the data of the **other** Queue object which is getting copied. *Note*: Consider why now you do need to implement this.
- **(4) Destructor** – will destroy the instance of the Queue object. *Note*: Consider why now you do need to implement this.
- **(5) operator=** will assign a new value to the calling Queue object, which will be an exact copy of the rhs object passed as a parameter. Returns a reference to the calling object to be used for cascading operator= as per standard practice. *Note*: Consider why now you do need to implement this.

- **(6a,6b) front** returns a Reference to the front element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.
- **(7a,7b) back** returns a Reference to the back element of the Queue. *Note*: Since it returns a Reference, before calling this method the user must ensure that the Queue is not empty.
- **(8) push** inserts at the back of the Queue an element of the given value. *Note*: No imposed maximum size limitations exist for the Node-based Queue variant.
- **(9) pop** removes from the front element of the Queue. *Note*: Checking if the Queue is empty prior to popping an element makes sense.
- **(10) size** will return the size of the current Queue.
- **(11) empty** will return a bool, true if the Queue is empty.
- **(12) full** will return a bool, true if the Queue is full. *Note*: Kept for compatibility, should always return false.
- **(13) clear** performs the necessary actions, so that after its call the Queue will be empty. *Note*: Consider now how you will implement this in contrast to the other queue variant.
- **(14) serialize** outputs to the parameter ostream os the complete content of the calling Queue object.

as well as a non-member overload for:

- **(i) operator<<** will output (to terminal or file) the complete content of the nodeQueue object passed as a parameter.

Final Note about Node-based Queue:

The required implementation will be a linear linked-list. This means that:

a) It will have a front pointer to the first element of the Queue, as seen in Lecture 20.
b) It will have a back pointer to the last element of the Queue, as seen in Lecture 20.
c) You may modify the specifications to add an auxiliary variable to keep track of the count of elements in the Queue (e.g. an size_t m_size) if you so wish.

You will create the necessary ArrayQueue.h and NodeQueue.h files that contain the necessary class declarations and implementations. You should also create a source file proj9.cpp which will be a test driver for your classes.

The completed project should have the following properties:

- Written, compiled and tested using Linux.
- It must compile successfully on the department machines using Makefile(s), which will be invoking the g++ compiler. Instructions how to remotely connect to department machines are included in the Projects folder in WebCampus.
- The code must be commented and indented properly.
  Header comments are required on all files and recommended for the rest of the program.
  Descriptions of functions commented properly.
- A one page (minimum) typed sheet documenting your code. This should include the overall purpose of the program, your design, problems (if any), and any changes you would make given more time.

Turn in: Compressed Header & Source files, Makefile(s), and project documentation.

Submission Instructions:

- You will submit your work via WebCampus
- Name your code file proj9.cpp
- If you have header file, name it proj9.h
- If you have class header and source files, name them as the respective class (ArrayQueue.h NodeQueue.h) This source code structure is not mandatory, but advised.
- Compress your:
  1. Source code
  2. Makefile(s)
  3. Documentation

  Do not include executable
- Name the compressed folder:
  PA#_Lastname_Firstname.zip
  Ex: PA9_Smith_John.zip

Verify: After you upload your .zip file, re-download it from WebCampus. Extract it, compile it and verify that it compiles and runs on the NoMachine virtual machines or directly on the ECC systems.

- Code that does not compile will be heavily penalized –may even cost you your *entire* grade–. Executables that do not work 100% will receive partial grade points.
- It is better to hand in code that compiles and performs partial functionality, rather than broken code. You may use your Documentation file to mention what you could not get to work exactly as you wanted in the given timeframe of the Project.

Late Submission:

A project submission is "late" if any of the submitted files are time-stamped after the due date and time. Projects will be accepted up to 24 hours late, with 20% penalty.